A COMPONENT-BASED DISTRIBUTED FRAMEWORK FOR WEB TRAVERSAL

Levente Hunyadi
MSc student
Budapest University of Technology and Economics
Department of Automation and Applied Informatics

Abstract

In web search engines, collecting source documents is an indispensable function to be performed periodically at high speeds. For scalable performance, parallelization is required, while flexibility is a must for easy adaptation to different demands. To meet both ends, an extensible, component-based, loosely-coupled distributed architecture for the .NET platform is presented. The architecture comprises of a lower layer that constitutes an execution environment and an upper layer that realizes a distributed crawler with a central coordinator.

1 INTRODUCTION

Since its birth in 1993, the Internet has undergone a tremendous change. In particular, it has dramatically increased in size and its content has reached an unprecedented diversity. As a result, locating scattered information has become a cumbersome endeavor. Hence, the role of automated *search engines* has increased substantially.

In order to compile a massive document index, search engines rely on *web crawlers*. Web crawlers traverse the Internet by hopping from document to document following *hyperlinks* they contain. For the index to remain up-to-date, web traversal should be performed efficiently. Single-machine architectures quickly stumble into system limits, which paves the way for parallelization. A distributed system executing simultaneously on multiple machines, however, involves a higher complexity in terms of communication, co-operation and synchronization.

Despite their complexity in management, distributed crawlers have major advantages in terms of scalability, dispersion of network load and an overall decrease in network traffic compared to centralized architectures. Computing and data storage capacity can adapt to the demands of the job by adjusting the number of workers (scalability), crawling agents can be placed at locations traffic-wise near the web domains they are to process (network load dispersion), thereby also requiring less data to be transmitted over the network (network traffic reduction). [4]

Even though web crawlers are widely employed, details of many of these systems are unknown or descriptions are too terse to allow reproducibility, partly to avoid malicious site creators deceiving engines to gain higher result rankings. In addition,

systems in published literature either have a centralized architecture [6] or do not lend themselves to easy configuration. [9, 2] A web crawler that harnesses the scalability potential in distributed systems whilst providing ample support for pluggable components and configurable behavior can act as a front-end for further research of search engines, web content or structure mining.

The proposed system is implemented in the .NET framework. The choice was driven by the versatility of this platform without a severe speed penalty. In addition, web crawlers realized as part of previous research are implemented in C, C++, Java, Perl or Python and no known distributed crawler is available for the .NET platform. The system is available for download through anonymous CVS login from the SourceForge.net site [1]. [7] gives an in-depth description of the proposed system.

2 ARCHITECTURAL OVERVIEW OF THE SYSTEM

In order to simultaneously exploit the merits of a distributed system and preserve a clear design open for extension, the proposed system is decomposed into two distinct layers. A *component framework* provides a standard solution to common tasks such as (possibly inter-process) communication, lifecycle management and configurability. In fact, the framework exposes an environment and abstract base classes that realize general behavior, the latter of which are extended with task-specific functionality by inheritor classes of the *crawler application*. Hence, the crawler application is developed as a set of loosely coupled units, wired together in a declarative (XML-based) manner to form a complete implementation.

The basic constituents of the underlying architectural layer are *components*, each of which is realized as a class executing a set of dedicated threads. Components are bound by means of local or remote *connectors* (realized as FIFO queues). Connectors transmit data between components and thus allow transparent distribution of components among possibly remote machines. *Providers*, the third major building block, offer machine-local services and access to persistent stores.

With the component architecture at our disposal, development of a full-fledged distributed web crawler is undertaken by extending the general component classes with crawler-oriented behavior such as scheduling download requests to hosts, downloading or parsing documents. The task-specific derived components are then interconnected to realize a distributed crawler architecture with a central coordinator. The coordinator (the *server*) is responsible for partitioning the web and assigning domains to agents (*clients*), which perform collecting and parsing pages in their domain, and maintaining the URL reference graph. Both the server and clients are highly configurable by replacing (or fine-tuning) the pluggable components they consist of.

3 THE COMPONENT FRAMEWORK

As previously mentioned, in spite of the scalability a distributed architecture provides, the complexity of the system increases sharply compared to a centralized system. Connection management, data transfer and error recovery become critical issues.

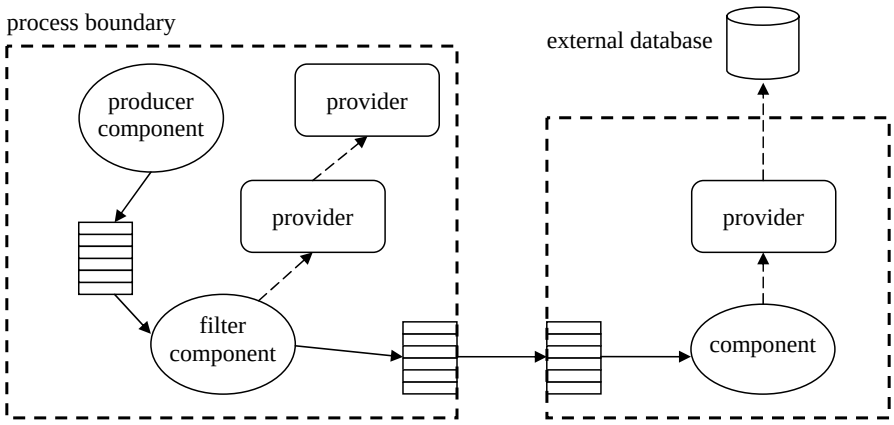

Figure 1: Interrelation of system building blocks

Nevertheless, these are external to the functional aspect of the crawling application. In the proposed system, the component framework shields the developer from the intricacy of the distributed architecture by providing an abstraction of components that consume input, perform action and produce output. For instance, a hyperlink extractor component receives documents and produces URLs contained in the document. Data flow between components is marshaled completely by the framework.

The overall architecture of the component framework resembles the *pipes and filters* pattern common to Unix-flavor operating systems. Three types of building blocks are distinguished. *Components* and *providers* encapsulate customizable functional behavior. *Connectors* are responsible for data exchange and buffering. (Figure 1)

Components are the primary constituents of the architecture. They generally consume, transform and produce data. For instance, a unit that verifies a URL against robot exclusion rules in effect for the given host and discards URLs that do not match constraints can be classified as a component. The abstract base class of components defines method stubs to override to realize user-defined functionality. Components may be synchronous or asynchronous depending on how they treat input and output. A *synchronous component* waits until all the input it requires is available, performs an operation based on all input and suspends until it manages to feed all data it has produced to its outputs. In contrast, an *asynchronous component* is activated by data on any of its inputs and free slots on any of its outputs. Obviously, asynchronous components are harder to code and require synchronization.

Providers offer access to external or machine-local shared data sources. For instance, a unit that executes a stored procedure through a database connection from a pool is a data provider, and a unit that downloads and caches robot exclusion files is a machine-local service provider. Providers may be bound to either components or other providers. The binding component or provider sees the bound provider through a well-defined interface. The bound provider is expected to give an implementation of the interface. For instance, a URL filter component could invoke a URL seen provider to check if a URL has previously been referenced. The provider interface exposes an `IsUrlSeen()` method that performs the test. The actual implementation of the provider queries an underlying database to see if the URL exists. On the other hand, if the actual implementation is replaced by a probabilistic structure, such as a Bloom

filter used by the Heritrix (Internet Archive) crawler [5], changes are required only in the configuration file. By changing the provider the component binds to, the system will run without recompilation. This approach also facilitates caching mechanisms.

Connectors are the glue between components and drive data flow in the distributed application. They are usually finite-capacity FIFO queues, which components may place items into or remote items from. Connectors can be either local or remote. A *local* connector stores references of data items and glues two components in the same process. A *remote* connector *serializes* and transmits data through a network.

4 CRAWLER APPLICATION

In the proposed crawler architecture, two participants are distinguished. A *server* is a central coordinator that assigns secondary (or higher level) domains to *clients*. Clients are the essence of the architecture which retrieve documents with respect to the appropriate traversal strategy and forward URLs back to the server they are not directly responsible for. Note that neither the server, nor clients are confined to a single or separate machines. In fact, the server and a client can be co-located. This is made possible by the component architecture that hides remote communication. Both server and client consist of multiple components.

Server components The *marshaler component* is the primary server component responsible for domain designation, thereby partitioning the web into disjoint parts. Coupled with a *communicator component* that maintains network connections to clients, the marshaler receives URLs that point out of client domains, verifies them against domain to client assignments and forwards the URL to the respective client. If no client has previously been selected for the domain, the marshaler chooses a client based on a (possibly random) selection algorithm. It may take into account the geographical location of the client and various global traversal constraints in effect (e.g. only to traverse the hu primary domain), all configurable through XML files and possibly the use of a different provider.

The volume of communication between clients and the server is limited. First, only 10% of page hyperlinks point out of their own domain. As a client can host multiple domains, even if the link is pointing out of the domain of the respective page, the URL may not have to be transmitted to the server. Furthermore, the reference graph of the web exposes a Zipfian pattern: a very small proportion of the total page set achieves high popularity and is referenced with outstanding frequency while most pages have one or two referencing links. Therefore, a most-recent caching scheme [3] with a capacity of 10 000 to 100 000 implemented in the marshaler component that discards recently referenced URLs significantly reduces server to client traffic. [4]

Client components The *traversal component* is the essence of each client and realizes the traversal strategy. Albeit it can be easily changed, its default behavior is to perform a *frequency-constrained breadth-first traversal* strategy. Here, URLs to be downloaded are appended to the tail of a disk-resident queue. A *load-balancer component*, working in cooperation with the traversal component, keeps track of hosts that have recently been polled and notifies the traversal component if any is ready for

Table 1: Comparison of the major characteristics of four crawler systems.

Criterion	Mercator	PolyBot	UbiCrawler	The proposed system
Architecture	centralized	distributed		
Configurability	high	medium	low	high
Extensibility	supported	limited	not supported	supported
Pluggability	fine-grained	coarse-grained	not supported	fine-grained
Target environment	single machine	LAN	unrestricted	
Scalability	limited		unlimited	
Primary limitation factor	disk speed	inter-component communication	suboptimal traversal	n.a.
Communication volume between participants	n.a.	large	low	medium
Default traversal	breadth-first	breadth-first	all pages on first host encounter	breadth-first
Host access control	single thread	data structure	single thread	data structure
Data storage	centralized		distributed	
Fault tolerance	medium	medium	highest	high
Language	Java	C++, Python	Java	.NET

another request, which serves to prevent overloading a specific server or domain. The traversal component extracts the URL that matches the host to which the notification refers and is closest to the head of the queue.

The URLs scheduled by the traversal component for retrieval are fetched by a set of *downloader components*. While the system is open for extension with various content-processing modules, tokenizers in particular, a hyperlink extractor *parser component* is of special interest. The links it finds in documents are forwarded to a URL *distributor component* that enforces robot exclusion rules [8] and transmits out-of-domain URLs to the server if necessary. Otherwise, links are appended to the previously mentioned URL queue.

5 COMPARISON TO OTHER SYSTEMS

In Table 1, the proposed crawler is compared to others with similar goals.

Mercator [6] is a scalable, extensible web crawler implemented in Java with a pluggable component architecture. Mercator is primarily centralized: it is a multi-threaded application, in which each individual *worker thread* executes independently. The scalability of the system is catered for by three factors: (1) multiple instances of a given component, possibly one for each worker thread, in bottlenecks; (2) a bounded memory use even for large data and (3) an efficient, random access-minimizing use of disk-resident structures to store data that does not fit into memory.

PolyBot [9] is a distributed web crawling architecture designed to run on a local network of workstations. It is implemented in C++ and Python. The system is built up of independent components each responsible for a well-defined task, while Network File System (NFS) provides a way to share data between these components. It is partitioned into two distinct parts: *crawling application* and *crawling system*. The customizable crawling application encapsulates the traversal strategy and forwards URLs to download to the crawling system. The crawling system, in contrast, is a more general part, which is responsible for tasks that are the same regardless of the

traversal method used, such as document retrieval, robot exclusion or DNS resolution.

UbiCrawler [2] is a fault-tolerant fully distributed web crawler implemented in Java. It consists of a set of independent, identically programmed *agents* that cooperate without a central coordinator. Agents communicate via Remote Method Invocation (RMI). Each agent is responsible for a set of hosts, that is determined by a one-way hash function of the domain name.

6 CONCLUSIONS AND PERSPECTIVES FOR FUTURE WORK

The proposed crawler realizes a dynamic partitioning of the web by the server assigning domains to clients. Clients traverse the domain they have been assigned and transmit URLs outbound from their domain back to the server. The system achieves limited client-server communication due to the small proportion of inter-domain compared to intra-domain URLs and global Zipfian distribution of document URLs.

The crawler is realized as a distributed architecture consisting of functional components. Component interconnection and message-based asynchronous communication is performed by an underlying framework. The framework exposes base classes, which component and provider developers can override to integrate new functionality into the system. The way components are wired to constitute a full-fledged crawler is specified deployment-time, which makes the system especially flexible compared to previous approaches.

The primary future work is to extend the system with a graphical user interface to monitor the crawl process and perform dynamic on-the-fly reconfiguration. Data storage is currently handled by a general-purpose database, which is to be replaced with dedicated storage structures as the application should satisfy no strict consistency but high speed criteria. Future work also includes further performance evaluation based on simulation and an actual large-scale web crawl.

References

[1] **SourceForge.net project site of Crawler.NET**, *http://sourceforge.net/projects/webcrawler*, February 1, 2007

[2] Paolo BOLDI, Bruno CODENOTTI, Massimo SANTINI, Sebastiano VIGNA, **UbiCrawler: A Scalable Fully Distributed Web Crawler**, *http://law.dsi.unimi.it/index.php*, February 1, 2007

[3] Andrei Z. BRODER, Marc NAJORK, Janet L. WIENER, **Efficient URL Caching for World Wide Web Crawling**, 2003

[4] Junghoo CHO, Hector GARCIA-MOLINA, **Parallel Crawlers**, WWW2002, Honolulu, Hawaii, May 7-11, 2002, ACM 1-58113-449-5/02/0005

[5] John Erik HALSE et al., **Heritrix developer documentation**, Internet Archive, *http://crawler.archive.org/articles/developer_manual/index.html*, January 24, 2007

[6] Allan HEYDON, Marc NAJORK, **Mercator: A Scalable, Extensible Web Crawler**, Compaq Systems Research Center, Palo Alto, CA

[7] HUNYADI Levente, PALLOS Péter, **Crawler.NET: A component-based distributed framework for web traversal**, Students' Scientific Conference (TDK) paper submission, Budapest University of Technology and Economics, Faculty of Electrical Engineering and Information Technology, November 17, 2006, *http://w3.enternet.hu/hunyadi/Publications/CrawlerNET/Report.pdf*, February 1, 2007

[8] Martijn KOSTER, **A Standard for Robot Exclusion**, June 30, 1994, *http://www.robotstxt.org/wc/exclusion.html*, August 6, 2006

[9] Vladislav SHKAPENYUK, Torsten SUEL, **Design and Implementation of a High-Performance Distributed Web Crawler**, Proceedings of the 18th International Conference on Data Engineering, CIS Department Polytechnic University, Brooklyn, NY, 2002